Paxter

Abhabongse Janthong

Jul 22, 2020

CONTENTS

1 Contents 3

2 Indices and tables 23

Index 25

Paxter is a document-first, text pre-processing mini-language toolchain, *loosely* inspired by @-expressions in Racket.

- The Paxter library package defines the syntax for **Paxter language** and provides a toolchain for parsing input texts written in Paxter language into *an intermediate parsed tree*.
- However, the semantics of Paxter language is left unspecified, meaning that users of the library have all the freedom to do whatever they like to render or transform the intermediate parsed tree into a final output they wish to achieve.
- Alternatively, instead of implementing an interpreter for intermediate parsed tree by themselves, users may opt-in to utilize a preset *parsed tree renderers*, also provided by this library package.

CHAPTER

ONE

CONTENTS

1.1 Getting Started

1.1.1 Installation

Paxter language package can be installed from PyPI via `pip` command (or any other methods of your choice):

```
$ pip install paxter
```

1.1.2 Programmatic Usage

This package is *mainly* intended to be utilized as a library. To get started, let's assume that we have a document source text written using **Paxter language syntax**.

```
# Of course, input text of a document may be read from any source,
# such as from a text file loaded from the filesystem, from user input, etc.

source_text = """\
@python##"
    from datetime import datetime

    name = "Ashley"
    year_of_birth = 1987
    current_age = datetime.now().year - year_of_birth
"##\\
My name is @name and my current age is @current_age.
My shop opens Monday@,-@,Friday.
"""
```

Note: Learn more about *Paxter language grammar and features*.

Parsing

First and foremost, we use a **parser** (implemented by the class *ParseContext*) to transform the source input into an intermediate parsed tree.

```
from paxter.core import ParseContext

parsed_tree = ParseContext(source_text).tree
```

Note: We can see the structure of the parsed tree in full by printing out its content as shown below (output reformatted for clarify).

```
>>> parsed_tree
FragmentList(
    start_pos=0,
    end_pos=236,
    children=[
        Command(
            start_pos=1,
            end_pos=148,
            starter="python",
            starter_enclosing=EnclosingPattern(left="", right=""),
            option=None,
            main_arg=Text(
                start_pos=10,
                end_pos=145,
                inner='\n    from datetime import datetime\n\n    name = "Ashley"\n
↪ year_of_birth = 1987\n    current_age = datetime.now().year - year_of_birth\n',
                enclosing=EnclosingPattern(left='##"', right='"##'),
            ),
        ),
        Text(
            start_pos=148,
            end_pos=161,
            inner="\\\nMy name is ",
            enclosing=EnclosingPattern(left="", right=""),
        ),
        Command(
            start_pos=162,
            end_pos=166,
            starter="name",
            starter_enclosing=EnclosingPattern(left="", right=""),
            option=None,
            main_arg=None,
        ),
        Text(
            start_pos=166,
            end_pos=189,
            inner=" and my current age is ",
            enclosing=EnclosingPattern(left="", right=""),
        ),
        Command(
            start_pos=190,
            end_pos=201,
            starter="current_age",
            starter_enclosing=EnclosingPattern(left="", right=""),
            option=None,
            main_arg=None,
```

(continues on next page)

(continued from previous page)

```
        ),
        Text(
            start_pos=201,
            end_pos=223,
            inner=".\nMy shop opens Monday",
            enclosing=EnclosingPattern(left="", right=""),
        ),
        SymbolCommand(start_pos=224, end_pos=225, symbol=","),
        Text(
            start_pos=225,
            end_pos=226,
            inner="-",
            enclosing=EnclosingPattern(left="", right=""),
        ),
        SymbolCommand(start_pos=227, end_pos=228, symbol=","),
        Text(
            start_pos=228,
            end_pos=236,
            inner="Friday.\n",
            enclosing=EnclosingPattern(left="", right=""),
        ),
    ],
    enclosing=GlobalEnclosingPattern(),
)
```

Notice how the source text above also contains what seems like a Python code. This has *nothing* to do with Paxter core grammar in any way; it simply uses the Paxter *command* syntax to *embed* Python code to which we will give a meaningful interpretation later.

Rendering

Next step, we use a built-in **renderer** to transform the intermediate parsed tree into its final output. It is important to remember that **the semantics of the documents depends on which renderer we are choosing**.

We will adopt the **Python authoring mode** whose renderer (implemented by `RenderContext`) is already predefined by the Paxter library package to transform the parsed tree into the desired final form. One of its very useful features is that it will execute python code under the `@python` command.

```
from paxter.pyauthor import RenderContext, create_unsafe_env

# This dictionary data represents the initial global dict state
# for the interpretation the document tree in python authoring mode.
env = create_unsafe_env({
    '_symbols_': {',': ' '},
})

result = RenderContext(source_text, env, parsed_tree).rendered
print(result)  # or write to a file, etc.
```

The above code will output the following.

```
My name is Ashley and my current age is 33.
My shop opens Monday - Friday.
```

Note: Learn more about *how to use Python authoring mode* and *how to write custom renderer*.

Create your own function

We recommend Paxter library users to by themselves write a utility function to connect all of the toolchains provided Paxter package. This is the minimal example of a function to get you started.

```
from paxter.core import ParseContext
from paxter.pyauthor import RenderContext, create_unsafe_env

def interp(source_text: str) -> str:
    parsed_tree = ParseContext(source_text).tree
    result = RenderContext(source_text, create_unsafe_env(), tree).rendered
    return result
```

1.1.3 Command-Line Usage

As a shortcut, Paxter library package also provided some utilities via command-line program. To get started, red the help message using the following command:

```
$ paxter --help
```

To play around with the parser, you may use `parse` subcommand with an input. Suppose that we have the following input file.

```
$ cat intro.paxter
@python##"
    from datetime import datetime

    _symbols_ = {
        ',': ' ',
    }
    name = "Ashley"
    year_of_birth = 1987
    current_age = datetime.now().year - year_of_birth
"##\
My name is @name and my current age is @current_age.
My shop opens Monday@,-@,Friday.
```

Then we can see the intermediate parsed tree using this command:

```
$ paxter parse -i intro.paxter
```

If we wish to also render the document written in Paxter language under the Python authoring mode with the default environment, then use the following command:

```
$ paxter pyauthor -i intro.paxter -o result.txt
$ cat result.txt
My name is Ashley and my current age is 33.
My shop opens Monday - Friday.
```

However, this command-line option does *not* provide a lot of flexibility. So we recommend users to dig deeper with a more programmatic usage. It may require a lot of time and effort to setup the entire toolchain, but it will definitely pay off in the long run.

1.2 Paxter Language Tutorial

Note: This is a tutotrial for *bare* Paxter language specification. It discusses only the basic Paxter syntax without any associated semantics as the semantics to the intermediate parsed tree is generally given by users of Paxter library.

For a simpler usage of Paxter library package, please also see *Python authoring mode tutorial page*.

Paxter syntax is very simple. In most cases, a typical text is a valid Paxter document, like in the following:

```
Hello, World!
My name is Ashley, and I am 33 years old.
```

However, Paxter provides a special syntax called **@-expressions** (pronounced "at expressions") so that richer information may be inserted into the document. There are 2 kinds of @-expressions, all of which begins with an @-symbol: a command and a short symbol expression.

This @-symbol (codepoint U+0040) is sometimes called a *switch* because it indicates the beginning of an @-expression, and whatever follows the switch determines which kind of @-expression it is.

Next, we dive into each kind of @-expressions.

Note: Consult *Syntax Reference* for a more detailed Paxter language grammar specification.

1.2.1 1. Command

A **command** is the most powerful syntax in Paxter language. It consists of the following 3 sections of information:

```
"@" starter [option] [main_argument]
```

Among these 3 sections, only the starter section is mandatory; the other 2 sections are optional and can be omitted. Additionally, there should *not* be any whitespace characters separating between the switch and the starter section, nor between different sections of the same command.

Starter section

A starter of a command may contain any textual content, surrounded by a pair of bars `|` (U+007C).

Here are examples of a valid command with only the starter section.

```
@|foo|
@|_create|
@||
@|foo.bar|
@|1 + 1|
@|Hello, World!|
```

However, if the content of the starter section takes the form of a valid Python identifier, then the pair of bars may be dropped. So the first 3 examples from above may be rewritten as follows:

```
@foo
@_create
@
```

On the other hand, the textual content of the starter may sometimes contain a bar as part of itself (such as `x || y || z`). Then we may additionally surround the matching pair of bars with an equal number of hashes `#` (U+0023):

```
@#|x || y || z|#
@###|x || y || z|###
```

But the following example will *not* work as expected:

```
@|x || y || z| is a command whose starter content contains exactly just "x "
followed by regular text "| y || z|".
```

Obviously, if the starter section begins with *n* hashes followed by a bar, then the textual content itself *cannot* contain a bar followed by *n* or more hashes (otherwise, the starter section would have terminated earlier).

```
@##|good|#|one|##
@##|bad|##|one|##
```

In this example (shown above), the starter of the first command is `good|#|boy` whereas that of the other command cuts short at `bad` (followed by the text `|one|##`).

Note: In a sense, this *bar pattern* (by which we mean the pattern of surrounding some content with a pair of bars plus an equal number of hashes on both ends) will be parsed **non-greedily** (i.e. the parsing of the starter halts as soon as the closing pattern corresponding to the opening pattern encountered earlier is found).

Main argument section

Let's skip the option section for now and discuss the main argument section of a command first.

As the name suggests, the main argument section of a command contains the most important piece of information to which the command is applied. The main argument can be supplied in one of 2 modes: the fragment list mode (in which the content is wrapped within the *brace pattern*) and the text mode (i.e. the content is wrapped within the *quoted pattern*).

(a) Wrapped fragment list mode

For a fragment list mode as the main argument, the content may contain texts as well as any *nested* @-expressions.

The content itself must be surrounded by a pair of curly braces (U+007B and U+007D) called the *brace pattern* (in analogous to the *bar pattern* associated with the starter section of a command). Of course, additionally appending the equal number of hashes to both ends are allowed.

For example,

```
@foo{Hello, @name}
@|font.large|{BUY ONE GET ONE FREE!}
@highlight##{A set of natural numbers: {0, 1, 2, 3, ...}.}##.
```

Similarly to the *bar pattern* from the starter section of a command, if the wrapped fragment list begins with *n* hashes followed by a left curly brace, then the **immediate** inner textual content may *not* contain a right curly brace followed by *n* or more hashes.

In the following example, the outermost command has the starter `foo` and its main argument is in fact `@bar{1###}###`. That is because (1) the curly braces pair surrounding `1###` (marked with "^") match with each other, and thus (2) the succeeding 3 hashes are not associated with the marked closing curly brace.

```
@foo###{@bar{1###}###}###
            ^    ^
```

(b) Wrapped text mode

Wrapped texts are somewhat similar to wrapped fragment lists, except for 2 major aspects:

- Instead of using a matching pair of curly braces surrounding the inner content, wrapped texts use a pair of quotation marks (U+0022). This is called the *quoted pattern* in analogous to the *brace pattern* for wrapped fragment lists.
- All @-symbol characters within the textual content will *not* be interpreted as the switch for @-expressions. Hence, wrapped texts would *not* contain any nested @-expressions.

This mode of main argument is useful especially when we expect the inner content of the main argument to be from **another domain** where @-symbols are prevalent.

For example, when you want to embed source code from another language:

```
@python_highlight##"

    # Results of the following function is cached
    # depending on its input
    from functools import lru_cache

    @lru_cache(maxsize=None)
    def add(x, y):
        """Adding function with caching."""
        return x + y

"##
```

Again, if the inner content needs to contain a quotation mark, we may add an equal number of hashes to both ends:

```
@alert#"Submit your feedback to "ashley@example.com"."#
```

Option section

The existence of a left square bracket immediately after the starter section of a command *always* indicates the beginning of the option section. The option section itself is a sequence of *tokens* where each token can be one of the following:

- **Another @-expression** of any kind
- **An identifier** (according to Python grammar)
- **An operator** which can be a single comma, a single semicolon, or a combination of all *other* symbol characters (excluding hashes, quotation marks, curly braces, and square brackets)
- **A number** whose syntactical form adheres to JSON grammar for number literal

- **A fragment list** wrapped within the *brace pattern* (which shares the same syntax as already discussed in the main argument section)
- **A text** wrapped within the *quoted pattern* (which shares the same syntax as already discussed in the main argument section)
- **A nested sequence of tokens** itself, surrounded by a matching pair of square brackets (U+005B and U+005D).

Warning: Please note that inside the option section of a command is the only place in Paxter language where whitespace characters between tokens are ignored.

Here are a couple of examples of commands which include the option section:

- For the command `@foo[x="bar", y=2.5, z={me}]{text}`, its option section contains a sequence of 11 tokens:
  1. an identifier `x`
  2. an equal sign operator `=`
  3. a text token `bar`
  4. a comma operator `,`
  5. an identifier `y`
  6. an equal sign operator `=`
  7. the number literal `2.5`
  8. a comma operator `,`
  9. an identifier `z`
  10. an equal sign operator `=`, and
  11. a fragment list containing the text `me`
- For the command `@|foo.bar|[x <- [2]; @baz]`, its option section contains a sequence of 5 tokens:
  1. an identifier `x`
  2. a left arrow operator `<-`
  3. a nested sequence containing the number literal `2` as the only token within it
  4. a semicolon operator `;`, and
  5. a nested command with `baz` as the starter section and with all other sections omitted.

Paxter language syntax gives a lot of freedom for what is allowed within the option section of a command; a programmer-write who writes a renderer to transform Paxter intermediate parsed trees into data of another form has a liberty to add whatever constraints to the syntactical structure within the option section.

1.2.2 2. Single Symbol Expression

This kind of @-expression is in the form of a single symbol character immediately following the @-symbol switch. This single *symbol* character will be the sole content of the single symbol expression.

For example,

```
There is free food today between 3@,-@,5 PM.
```

Warning: If `@#` happens to be the prefix of a full-form @-expressions (such as in `@#|foo|#`), then `@#` by itself is *not* a valid command in special form. It must be **unambiguously** *not* part of full-form command for itself to become a valid command of special form.

1.2.3 Escaping @-Symbol Switches

Paxter language does *not* provide any syntax to escape @-symbol switches of @-expressions. We recommend the library user solve this kind of problem at the interpreter/renderer level instead.

One way to do this is to define the behavior of `@@` (a single symbol expression with `@` symbol following the switch) to be transformed into a single `@` symbol in the rendered output.

```
My email is ashley@@example.com.
```

Another method to work around this problem is to introduce a command called `verbatim` (inspired by the command of the same name in LaTeX) which will output the main input argument as-is.

```
My email is @verbatim"ashley@example.com".
```

1.3 Python Authoring Mode Tutorial

1.3.1 Block Python Code Execution

In Python authoring mode, Python source code may be embedded into the document for execution using `python` command syntax with the code as the main argument. For example,

```
@python##"
    name = "Ashley"
"##
```

In the example document above, once the Python code in the preamble is executed, the value of the variable `name` will be available in the environment for the rest of the document.

Referring to variable from Python code

One way to referring to the value of the variable `name` is to use the command syntax `@name` without any options or main arguments sections. So the following document

```
@python##"
    name = "Ashley"
"##
Hi, @name.
```

will be rendered into

```

Hi, Ashley.
```

Remove unwanted newlines

Notice how the newline character was preserved in the above output. If we wish to remove that newline character, we may put a backslash at the end of that line. So the following document

```
@python##"
    name = "Ashley"
"##\
Hi, @name.
```

yields the following output in Python authoring mode

```
Hi, Ashley.
```

Referring to functions from Python code

We may also define Python functions within the embedded Python source code and refer to them later in the document. The syntax to make a call to a function already defined is a command syntax with the main argument supplied. Here is one example,

```
@python##"
    def surround(text):
        return "(" + flatten(text) + ")"
"##\
This is @surround{sound}.
```

which will return

```
This is (sound).
```

The reason why we need to `flatten` the main argument first is that the fragment list (i.e. the part surrounded by a matching pair of curly braces) returns a list of string tokens (not the string itself), hence it is important to flatten them into a single string first (otherwise an error would have occurred).

Python functions with multiple arguments

When there is more than one argument to the function, the main argument of the command will always be the first argument of the function, and the rest of the function arguments can be supplied to option section of the command (similarly to Python function call syntax):

```
@python##"
    def surround(text, n, left='(', right=')'):
        return flatten(left) * n + flatten(text) + flatten(right) * n
"##\
This is @surround[3]{sound}.
This is @surround[n=3]{sound}.
This is @surround[3, "[", "]"]{sound}.
This is @surround[3, right=""]{sound}.
This is @surround[n=3, left="_", right="_"]{sound}.
```

Here is the result.

```
This is (((sound))).
This is (((sound))).
This is [[[sound]]].
This is (((sound.
This is ___sound___.
```

Notice that we use wrapped text inside the option section in order to supply strings as arguments to the function `surround`.

Additionally, we may also omit the main argument section, and then the entire option section will all be the arguments to the function:

```
@python##"
    def surround(text, n, left='(', right=')'):
        return flatten(left) * n + flatten(text) + flatten(right) * n
"##\
This is @surround["sound",3].
This is @surround["sound",n=3].
```

The above document will be rendered into

```
This is (((sound))).
This is (((sound))).
```

1.3.2 Inline Python Code Evaluation

We may wish to insert the result of the evaluation of Python expression. We can do so by using the command syntax with the bar pattern `@|...|`:

```
The result of 7 × 11 × 13 is @|7 * 11 * 13|.
```

and that would be transformed into

```
The result of 7 × 11 × 13 is 1001.
```

Inline Python code with function call

If a function behind an attribute or key lookup, we may use the bar pattern in conjunction with main arguments and/or options.

```
@python##"
    import statistics
    values = [2, 3, 5, 7]
    funcs = {
        'median': statistics.median
    }
"##\
The average of first 4 primes is @|statistics.mean|[@values].
The median of first 4 primes is @|funcs['median']|[@values].
```

The above document returns the following.

```
The average of first 4 primes is 4.25.
The median of first 4 primes is 4.0.
```

1.3.3 Special Symbol Commands

For the sake of simplicity, we provide an easy way to perform text replacements for symbol-style commands. Simply define a dictionary mapping from each symbol to the substituting results under the variable `_symbol_` inside the Python source code.

```
@python##"
    _symbols_ = {
        '.': ' ',
        ',': ' ',
        '@': '@',
    }
"##\
My email is ashley@@example.com.
My office hours is between 7@.-@.9 PM.
```

Here is the result of the above document.

```
My email is ashley@example.com.
My office hours is between 7 - 9 PM.
```

1.3.4 Special Commands: For and If

For statements within the document for Python authoring mode has the following format

```
@for[<IDENTIFIER> in <EXPRESSION>]{<BODY>}
```

whereas if statements has the 3 following formats

```
@if[<CONDITIONAL>]{<BODY>}
@if[not <CONDITIONAL>]{<BODY>}
@if[<CONDITIONAL> then <THEN_BODY> else <ELSE_BODY>]
```

Here is the document that illustrates how to use these special commands:

```
@python##"
    def is_odd(value):
        return value % 2 == 1
"##\
Odd digits are @flatten{@for[i in @|range(10)|]{@if[@|is_odd(i)|]{ @i}}}.
Even digits are @flatten{@for[i in @|range(10)|]{@if[not @|is_odd(i)|]{ @i}}}.
Digits are @flatten{@for[i in @|range(10)|]{@if[@|is_odd(i)| then " odd" else " even
↪"]}} in this order.
```

and the result would be

```
Odd digits are 1 3 5 7 9.
Even digits are 0 2 4 6 8.
Digits are  even odd even odd even odd even odd even odd in this order.
```

1.3.5 API Reference

The following class implements a standard parser which comes with Paxter package library.

The following function creates a pre-defined unsafe Python environment dictionary to be used with the rendering context class.

Here are the functions readily available within the default environment from the function above

`paxter.pyauthor.funcs.`**`flatten`**(*data*, *is_joined: bool = True*) → Union[List[str], str]

Flattens the nested list of elements by unrolling them into a single list. Unless the `is_joined` option is disabled, all elements will be combined to a single string.

```
>>> flatten(["Hello", ",", " ", "World", "!"])
"Hello, World!"
>>> flatten(["Hello", [",", " "], ["World"], "!"])
"Hello, World!"
>>> flatten(["Hello", [",", " "], ["World"], "!"], is_joined=False)
["Hello", ",", " ", "World", "!"]
>>> flatten("Hello, World!")
"Hello, World!"
>>> flatten("Hello, World!", is_joined=False)
["Hello, World!"]
```

`paxter.pyauthor.funcs.`**`verb`**(*text: Any*) → str

Returns the main string argument as-is.

```
>>> verb("Hello")
"Hello"
>>> verb("me@example.com")
"me@example.com"
```

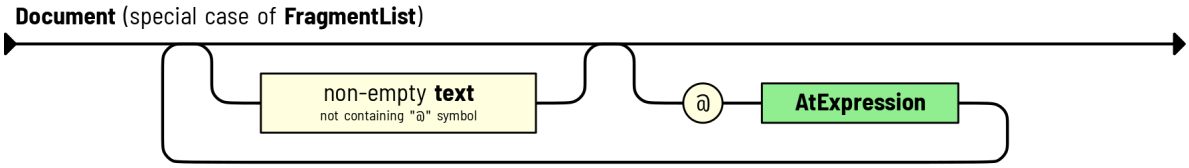
1.4 Custom Renderer Tutorial

Todo: Tutorial is coming soon.

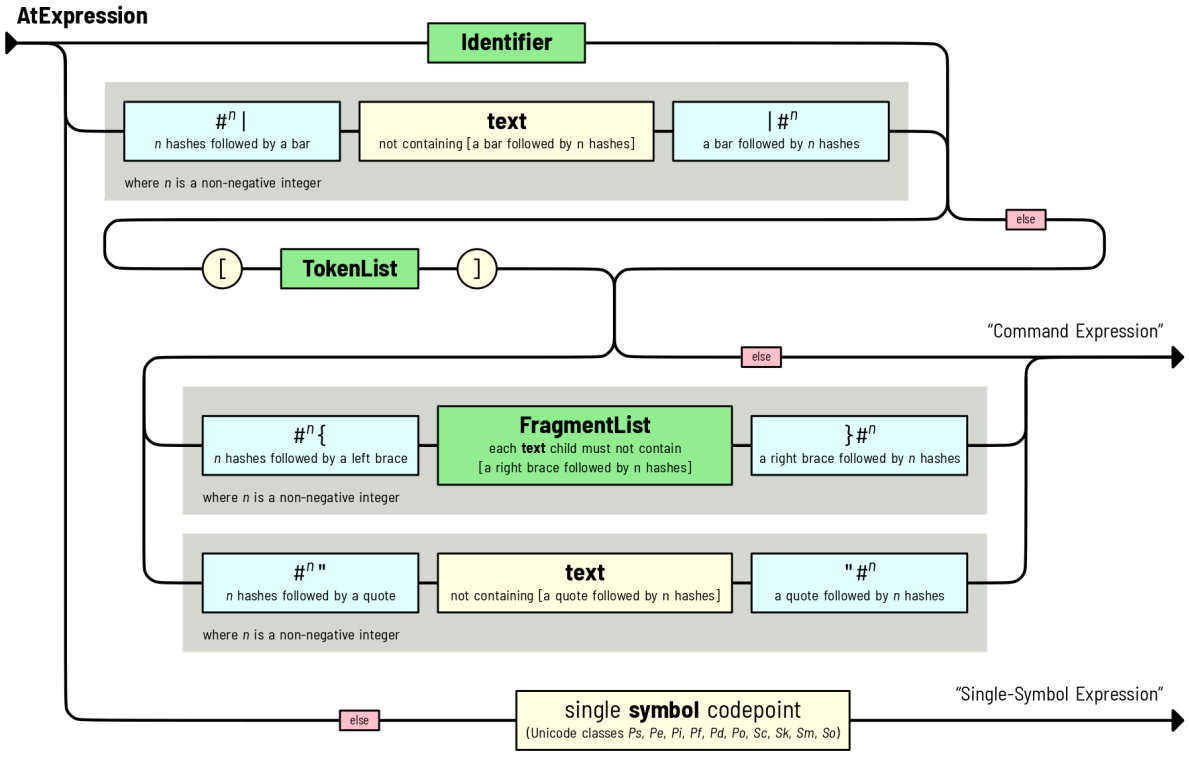
1.5 Syntax Reference

Below are syntax diagrams for Paxter language.

- **Document:** Starting rule of Paxter language grammar. It is a special case of **FragmentList** rule, and thus the result is always a `FragmentList` node whose children are non-empty `Text` interleaving with the result produced by **AtExpression** rule.



- **AtExpression:** Rule for parsing right after encountering @-switch symbol.



Note: The red `else` box in this diagram indicates that such path can be followed only if the next token does not match any other possible paths. Pursuing this `else` path does not consume anything.

There are 2 possible scenarios.

1. A normal *`Command`* node consisting of 3 sections: starter, options, and main argument, respectively.

   The starter section is resulted from parsing either greedily for an identifier or non-greedily for a text enclosed by a pair of bars plus and an equal number of zero or more hashes at both ends.

   Following the starter section, if a left square bracket is found, then the option section as a list of tokens must be parsed and it will result in a *`TokenList`* node. Otherwise (if the left square bracket is absent), this option section will be represented by `None`.

   Finally, the main argument section. (a) If there is zero or more hashes followed by a left brace, then the **FragmentList** parse rule must be followed and thus yields *`FragmentList`* as the result.

   > **Warning:** There is a restriction imposed on parsing the **FragmentList** rule, which is that the child text node may not contain a right brace followed by the same number of hashes as the preceding part. Otherwise, the parsing of **FragmentList** rule would have terminated earlier.

   However, (b) if there is zero or more hashes followed by a quotation mark, then the text is parsed non-greedily until the another quotation mark followed by the same number of hashes is found.

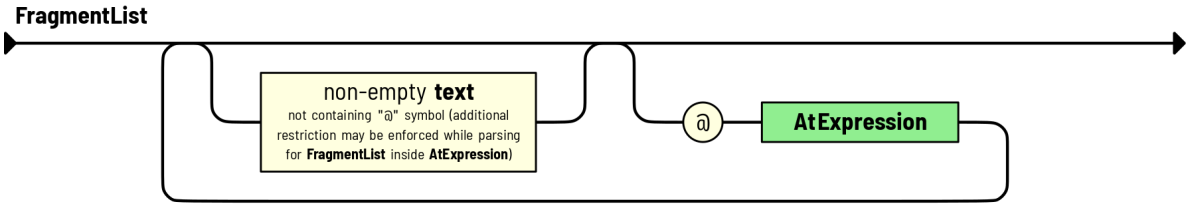
   Well, if both conditions (a) and (b) do not hold, then the main argument would be `None`.

2. A special `SingleSymbol` node where a single symbol follows the `@`-switch.

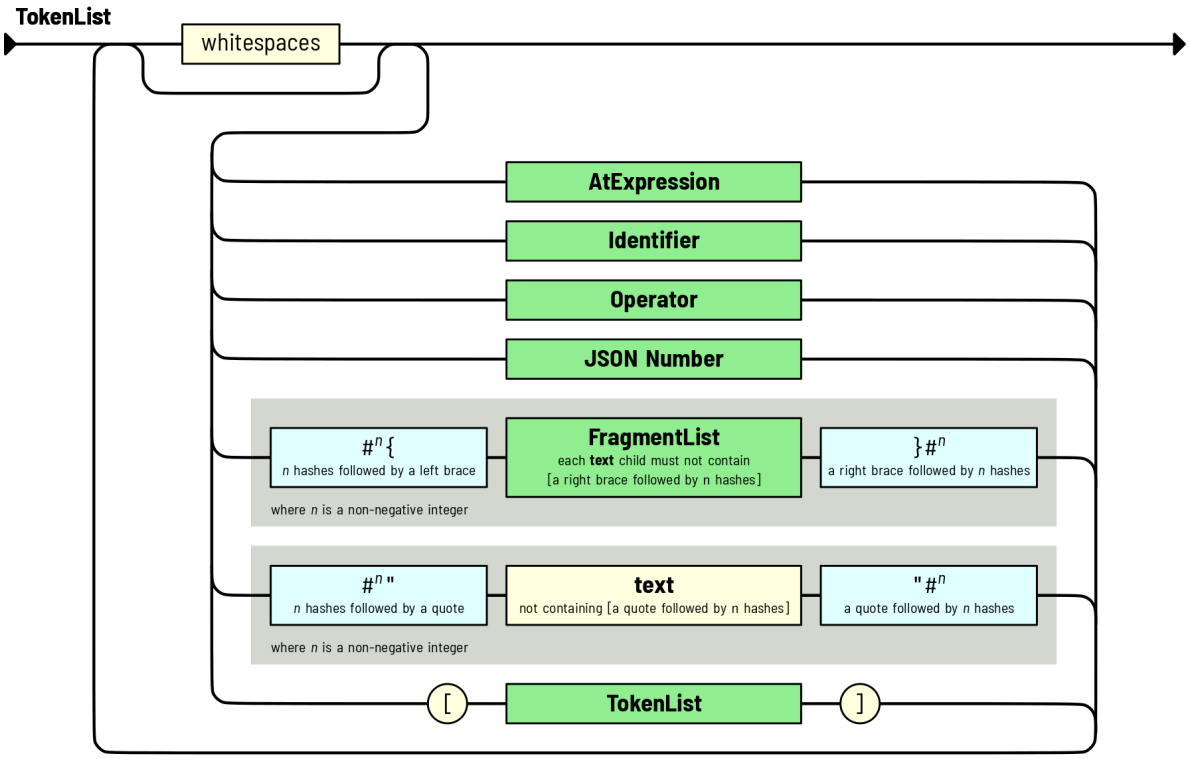
- **FragmentList:** Consists of an interleaving of non-empty texts and results produced by **AtExpression** rule.

  Note that the parsing of **AtExpression** rule at the *previous level* may put some restriction on the parsing of *`Text`* nodes. For example, if preceding the fragment list is an opening brace pattern `###{`, then each *`Text`* node may contain `}###`.

  In other words, we *non-greedily* parses text within the fragment list.

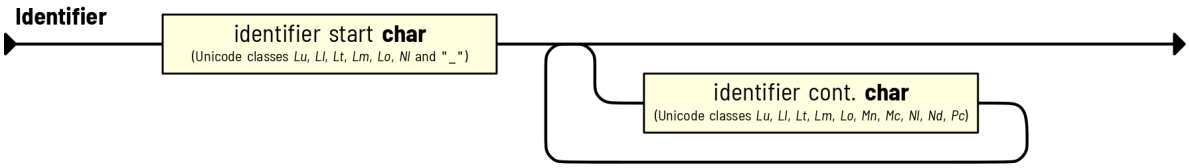


- **TokenList:** A sequence of zero or more tokens Each token either a command, an identifier, an operator, a number following JSON specification, a wrapped fragment list, a wrapped text, or a nested token list enclosed by a pair of square brackets `[]`. The result is a *`TokenList`* node type.

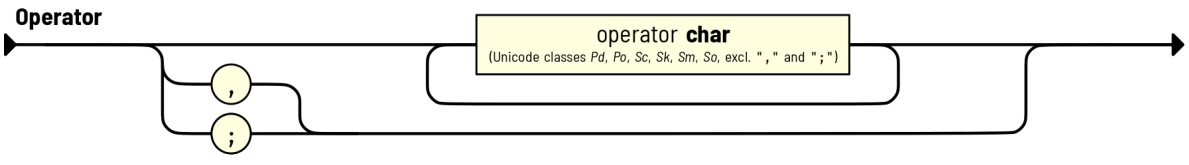


Note: The option section (or the token list) is the only place where whitespaces are ignored (when they appear between tokens).

- **Identifier:** Generally follows Python rules for greedily parsing an identifier token (with some extreme exceptions). The result is an `Identifier` node type.



- **Operator:** Greedily consumes as many operator character as possible (with two notable exceptions: a comma and a semicolon, which has to appear on their own). A whitespace may be needed to separate two consecutive, multi-character operator tokens. The result is an `Operator` node type.



1.6 Core API Reference

Paxter language package provides the following core functionality.

1.6.1 Parsing

This class implements the parser for Paxter language.

class paxter.core.**ParseContext**(*input_text: str*)
Implements a recursive descent parser for Paxter language text input.

To utilize this class, provide the input text to the constructor, and the resulting parsed tree node will be generated upon instantiation.

input_text: str
Document source text

tree: FragmentList
Root node of the parsed tree

1.6.2 Data Definitions

The result of the parsing yields the parsed tree consisting of the following classes.

class paxter.core.**Token**(*start_pos: int*, *end_pos: int*)
Base class for all types of nodes to appear in Paxter document tree.

end_pos: int
The index right after the ending position of the token

start_pos: int
The index of the starting position of the token

class paxter.core.**Fragment**(*start_pos: int*, *end_pos: int*)
Bases: paxter.core.data.Token

Subtypes of nodes in Paxter document tree that is allowed to appear as direct members of *FragmentList*.

class paxter.core.**TokenList**(*start_pos: int*, *end_pos: int*, *children: List[paxter.core.data.Token]*)
Bases: paxter.core.data.Token

Node type which represents a sequence of tokens wrapped under a matching pair of brackets `[]`, all of which appears only within the option section of *Command*.

children: List[Token]
List of *Token* instances

class paxter.core.**Identifier**(*start_pos: int*, *end_pos: int*, *name: str*)
Bases: paxter.core.data.Token

Node type which represents an identifier, which can appear only within the option section of *Command*.

name: str
Identifier string name

class paxter.core.**Operator**(*start_pos: int*, *end_pos: int*, *symbols: str*)
Bases: paxter.core.data.Token

Node type which represents an operator, which can appear only within the option section of *Command*.

symbols: str
Symbol as a string of characters

class paxter.core.**Number**(*start_pos: int*, *end_pos: int*, *value: Union[int, float]*)
Bases: paxter.core.data.Token

Node type which represents a number recognized by JSON grammar, which can appear only within the option section of *Command*.

value: Union[int, float]
Numerical value deserialized from the number literal

class paxter.core.**FragmentList**(*start_pos: int*, *end_pos: int*, *children: List[paxter.core.data.Fragment]*, *enclosing: paxter.core.enclosing.EnclosingPattern*)
Bases: paxter.core.data.Token

Special intermediate node maintaining a list of fragment children nodes. Nodes of this type usually correspond to either the global-level fragments or fragments nested within enclosing brace pattern.

The enclosing brace pattern may appear as the main argument of a *Command* node or as a token within the option section of a *Command* node.

children: List[Fragment]
List of *Fragment* instances

enclosing: EnclosingPattern
Information of the enclosing braces pattern

class paxter.core.**Text**(*start_pos: int*, *end_pos: int*, *inner: str*, *enclosing: paxter.core.enclosing.EnclosingPattern*)
Bases: paxter.core.data.Fragment

Text node type which does not contain nested @-expressions. Nodes of this type usually be presented as an element of *FragmentList* or as text wrapped within enclosing quoted pattern.

The enclosing quote pattern may appear as the main argument of a *Command* node, as a token within the option section of a *Command* node, or as a fragment element of a *FragmentList* node.

enclosing: EnclosingPattern
Information of the enclosing quote pattern

inner: str
Inner string content

class paxter.core.**Command**(*start_pos: int*, *end_pos: int*, *starter: str*, *starter_enclosing: paxter.core.enclosing.EnclosingPattern*, *option: Optional[paxter.core.data.TokenList]*, *main_arg: Optional[Union[FragmentList, Text]]*)
Bases: paxter.core.data.Fragment

Node type representing @-expression which has the following form:

- It begins with an `@` switch character.
- Then, it is immediately followed by a section called a starter which is simply a string in valid Python identifier form or a string surrounded by enclosing bar pattern: `|...|`.
- Next, it may optionally be followed by an option section which is a sequence of *Token* nodes.
- Finally, it may optionally be followed by a main argument section which can either be a *FragmentList* or a *Text*.

main_arg: Optional[MainArgument]
The main argument section at the end of expression, or `None` if this section is not present.

option: Optional[TokenList]
A list of tokens for the option section enclosed by `[]`, or `None` if this section is not present.

starter: str
Command starter section

starter_enclosing: EnclosingPattern
Information of the enclosing bar pattern over the starter section

1.6.3 Exceptions

Here are the list of exceptions raised from this library.

class paxter.core.exceptions.**PaxterBaseException**(*message: str*, ***positions: paxter.core.charloc.CharLoc*)
Bases: `Exception`

Base exception specific to Paxter language ecosystem.

message: str
Error message

positions: Dict[str, CharLoc]
A mapping from position name to `LineCol` position data

class paxter.core.exceptions.**PaxterConfigError**(*message: str*, ***positions: paxter.core.charloc.CharLoc*)
Bases: *paxter.core.exceptions.PaxterBaseException*

Exception for configuration error.

class paxter.core.exceptions.**PaxterSyntaxError**(*message: str*, ***positions: paxter.core.charloc.CharLoc*)
Bases: *paxter.core.exceptions.PaxterBaseException*

Exception for syntax error raised while parsing input text in Paxter language. Positional index parameters indicates a mapping from position name to its indexing inside the input text.

class paxter.core.exceptions.**PaxterRenderError**(*message: str*, ***positions: paxter.core.charloc.CharLoc*)
Bases: *paxter.core.exceptions.PaxterBaseException*

Exception for parsed tree transformation error.

1.6.4 Other Utility Classes

Classes in this subsection is for reference only.

class paxter.core.**EnclosingPattern**(*left: str*, *right: str = None*)
Information regarding the enclosing (left and right) patterns for a particular scope of string data.

left: str
The left (i.e. opening) pattern enclosing the scope

right: str = None
The right (i.e. closing) pattern enclosing the scope

class paxter.core.**GlobalEnclosingPattern**
Specialized scope pattern just for global-level fragment list.

class paxter.core.**CharLoc**(*input_text: dataclasses.InitVar*, *pos: dataclasses.InitVar*)

The position (starting or ending) of a token within the input text useful for line and column information in error messages.

col: int

1-index column index value

line: int

1-index line number

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

INDEX

C

CharLoc (*class in paxter.core*), 21
children (*paxter.core.FragmentList attribute*), 20
children (*paxter.core.TokenList attribute*), 19
col (*paxter.core.CharLoc attribute*), 22
Command (*class in paxter.core*), 20

E

enclosing (*paxter.core.FragmentList attribute*), 20
enclosing (*paxter.core.Text attribute*), 20
EnclosingPattern (*class in paxter.core*), 21
end_pos (*paxter.core.Token attribute*), 19

F

flatten() (*in module paxter.pyauthor.funcs*), 15
Fragment (*class in paxter.core*), 19
FragmentList (*class in paxter.core*), 20

G

GlobalEnclosingPattern (*class in paxter.core*), 21

I

Identifier (*class in paxter.core*), 19
inner (*paxter.core.Text attribute*), 20
input_text (*paxter.core.ParseContext attribute*), 19

L

left (*paxter.core.EnclosingPattern attribute*), 21
line (*paxter.core.CharLoc attribute*), 22

M

main_arg (*paxter.core.Command attribute*), 20
message (*paxter.core.exceptions.PaxterBaseException attribute*), 21

N

name (*paxter.core.Identifier attribute*), 19
Number (*class in paxter.core*), 20

O

Operator (*class in paxter.core*), 19
option (*paxter.core.Command attribute*), 21

P

ParseContext (*class in paxter.core*), 19
PaxterBaseException (*class in paxter.core.exceptions*), 21
PaxterConfigError (*class in paxter.core.exceptions*), 21
PaxterRenderError (*class in paxter.core.exceptions*), 21
PaxterSyntaxError (*class in paxter.core.exceptions*), 21
positions (*paxter.core.exceptions.PaxterBaseException attribute*), 21

R

right (*paxter.core.EnclosingPattern attribute*), 21

S

start_pos (*paxter.core.Token attribute*), 19
starter (*paxter.core.Command attribute*), 21
starter_enclosing (*paxter.core.Command attribute*), 21
symbols (*paxter.core.Operator attribute*), 19

T

Text (*class in paxter.core*), 20
Token (*class in paxter.core*), 19
TokenList (*class in paxter.core*), 19
tree (*paxter.core.ParseContext attribute*), 19

V

value (*paxter.core.Number attribute*), 20
verb() (*in module paxter.pyauthor.funcs*), 15